Today

- Today we're talking computer architecture… from a software engineer's perspective

- Key concepts about how modern parallel processors achieve high throughput
  - Two concern parallel execution (multi-core, SIMD parallel execution)
  - One addresses the challenges of memory latency (multi-threading)

- Understanding these basics will help you
  - Understand and optimize the performance of your parallel programs
  - Gain intuition about what workloads might benefit from fast parallel machines

Compile program

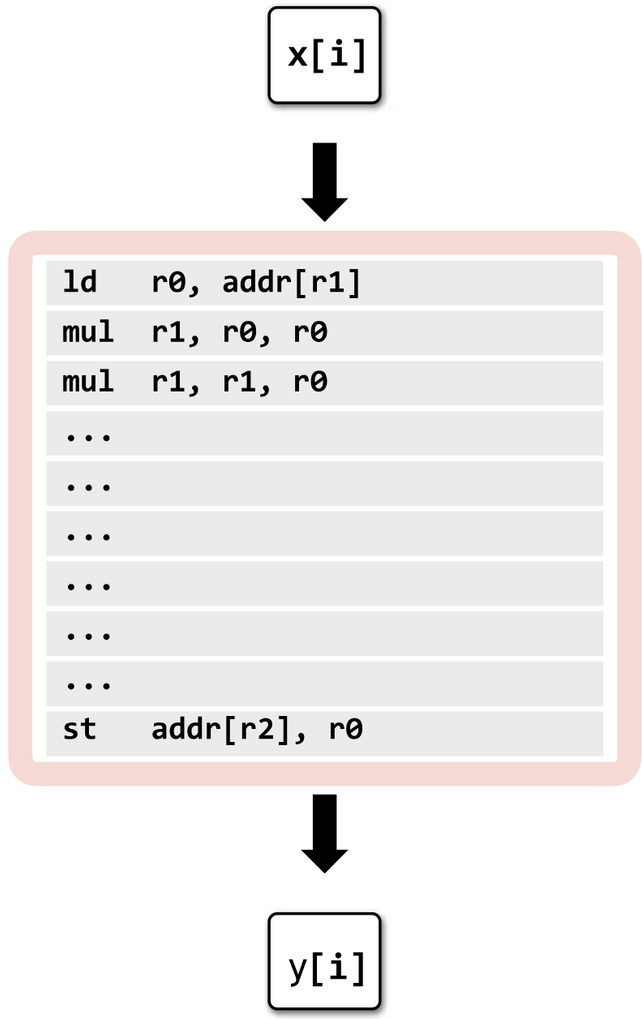
```
void sinx(int N, int terms, float* x, float* y)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      y[i] = value;
   }
}
```

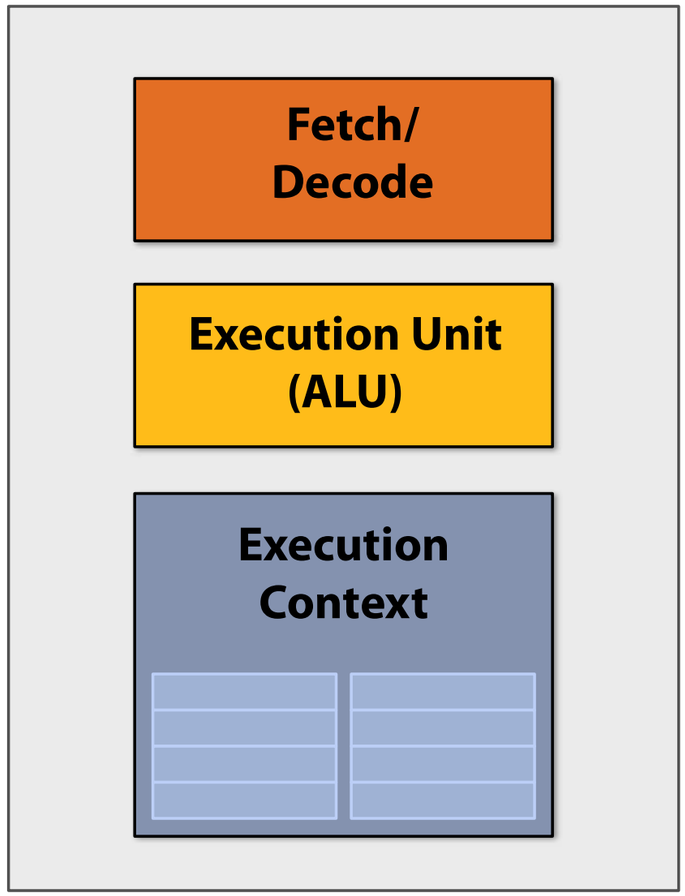
compiler

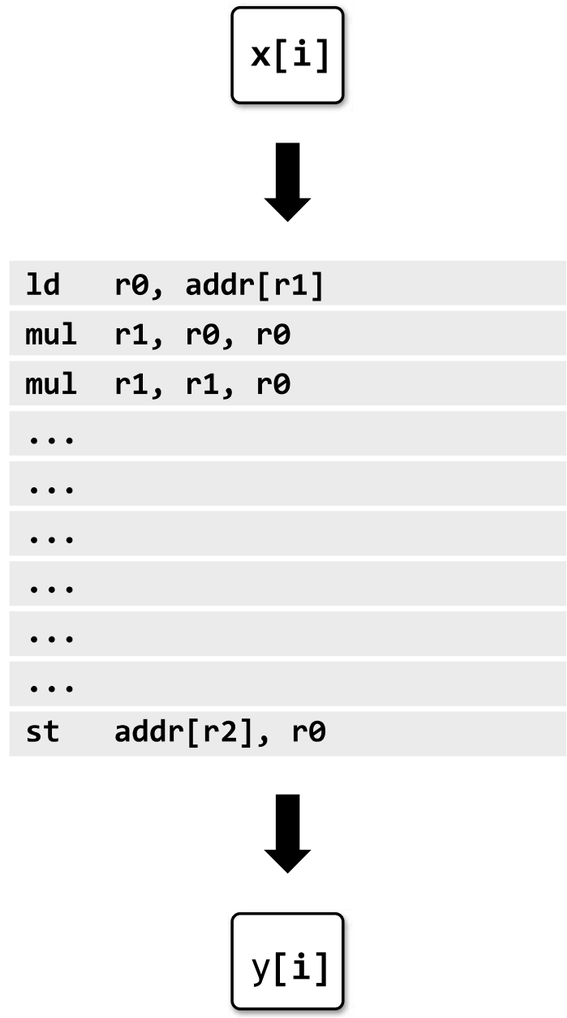
Compiled instruction stream (scalar instructions)



Execute program

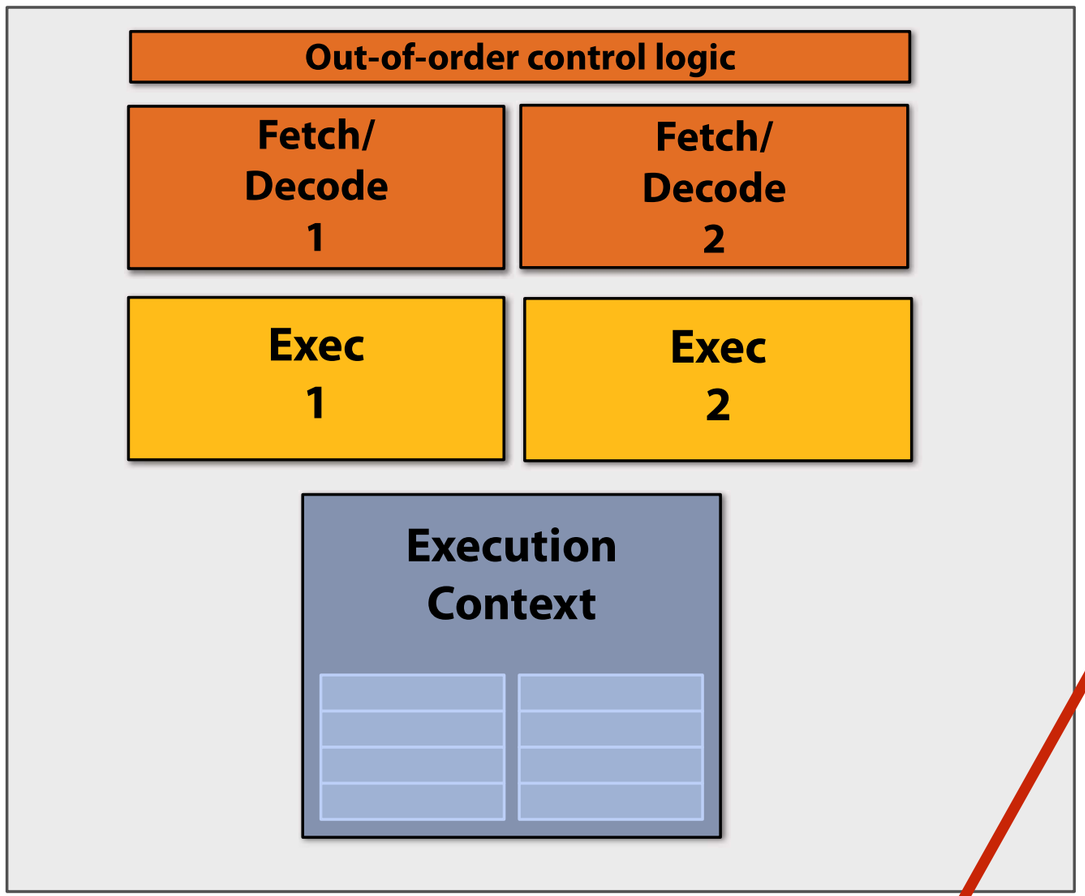
My very simple processor: executes one instruction per clock

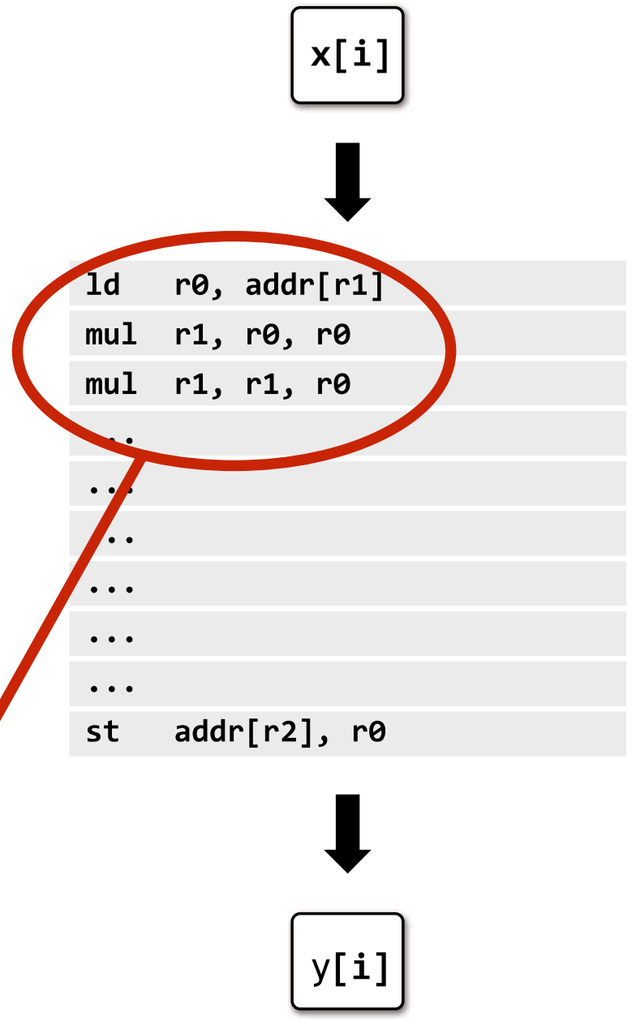




Superscalar processor

The processor shown here can decode and execute two instructions per clock (if independent instructions exist in an instruction stream)

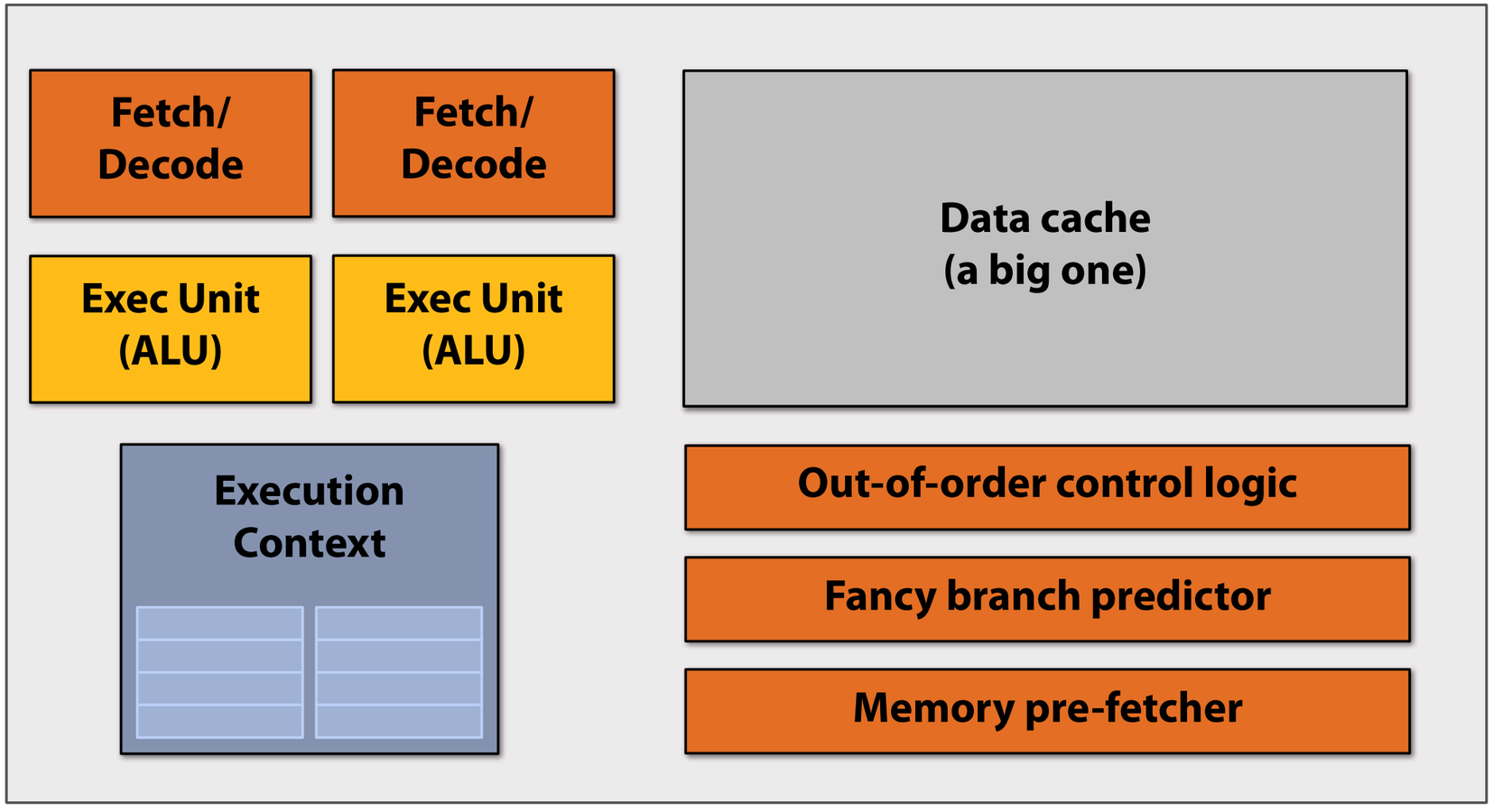




Note: No ILP exists in this region of the program

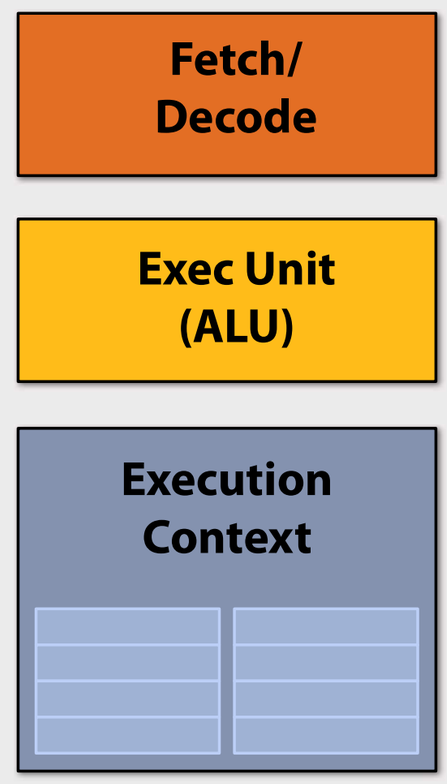
Pre multi-core era processor

Majority of chip transistors used to perform operations that help make a single instruction stream run fast



More transistors = larger cache, smarter out-of-order logic, smarter branch predictor, etc.

Multi-core era processor

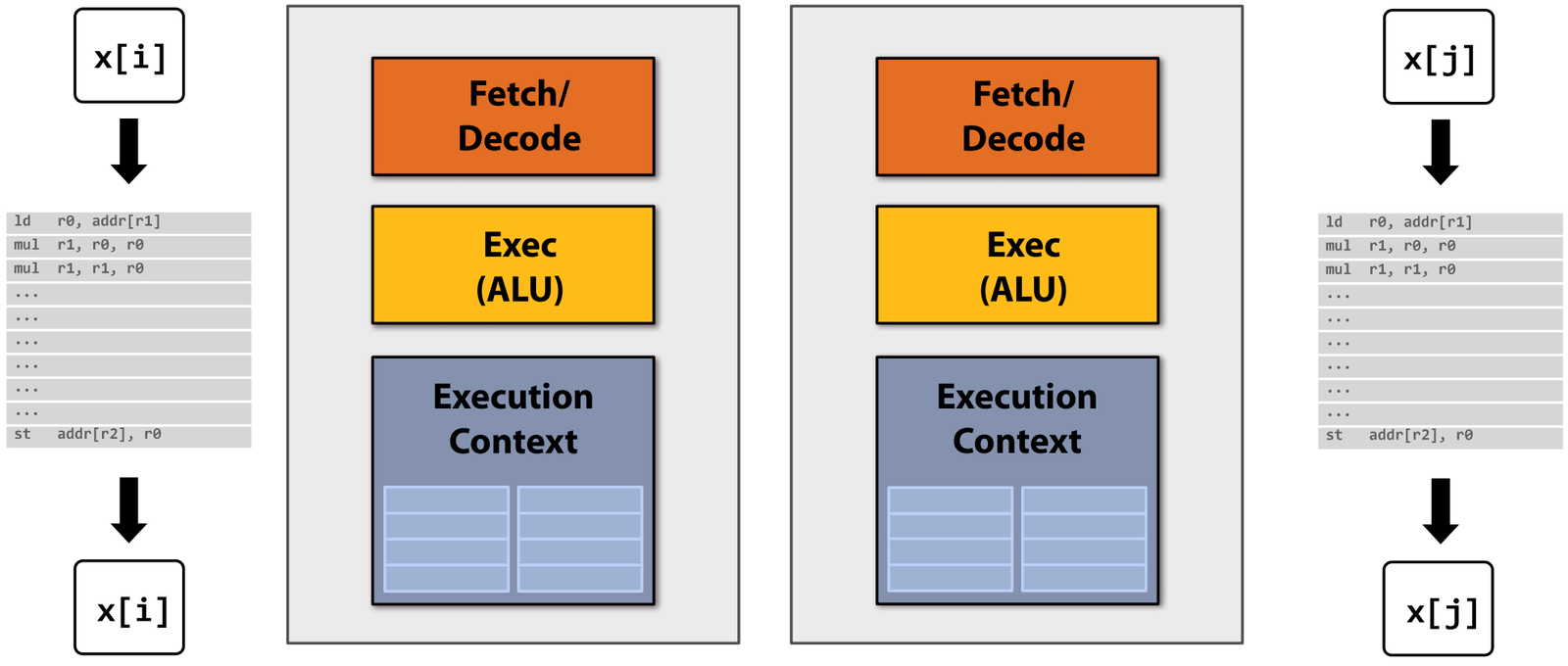


Idea #1:

Rather than use transistors to increase sophistication of processor logic that accelerates a single instruction stream (e.g., out-of-order and speculative operations)

Use increasing transistor count to add more cores to the processor

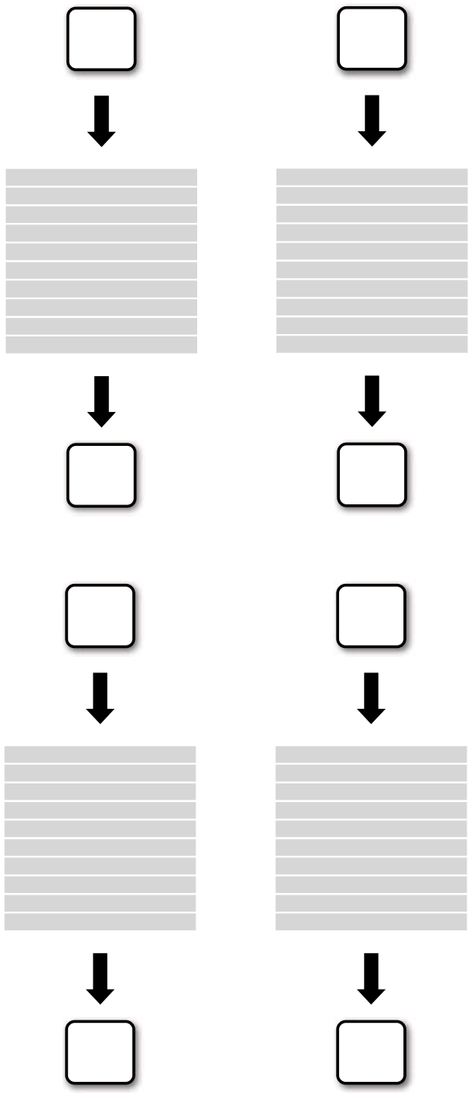
Two cores: compute two elements in parallel

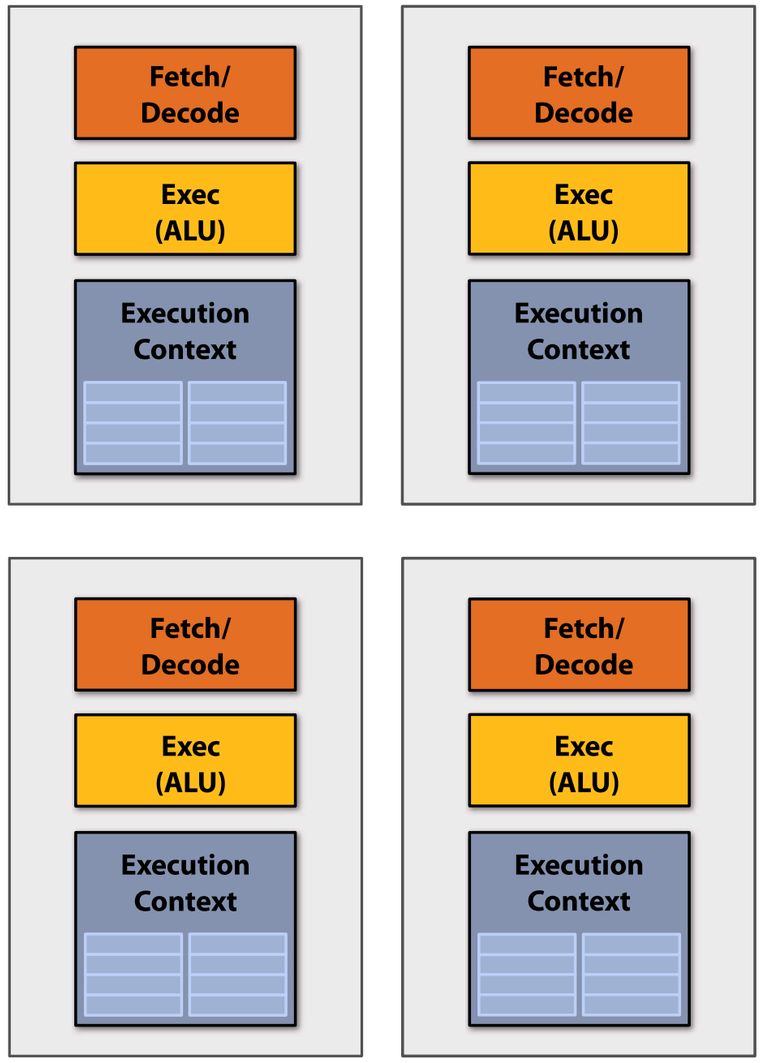


Simpler cores: each core may be slower at running a single instruction stream than our original "fancy" core (e.g., 25% slower)

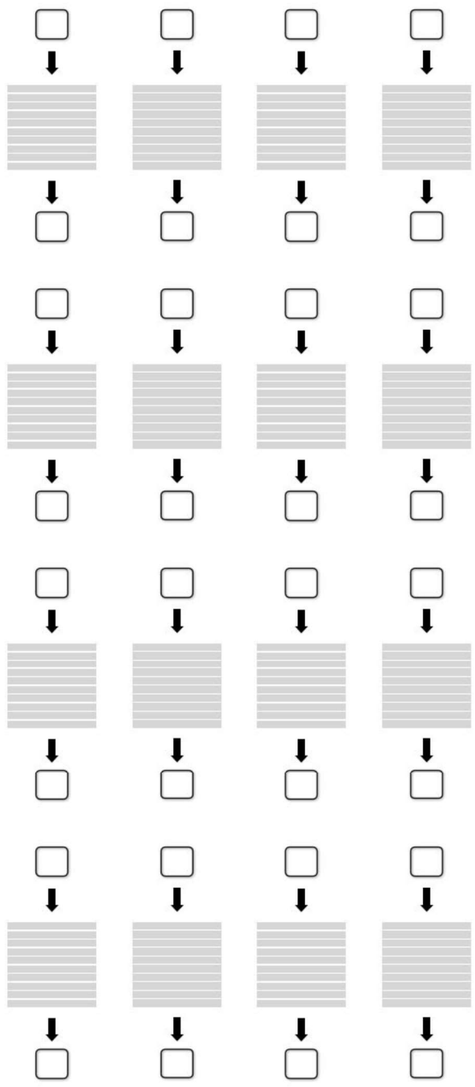
But there are now two cores: $2 \times 0.75 = 1.5$ (potential for speedup!)

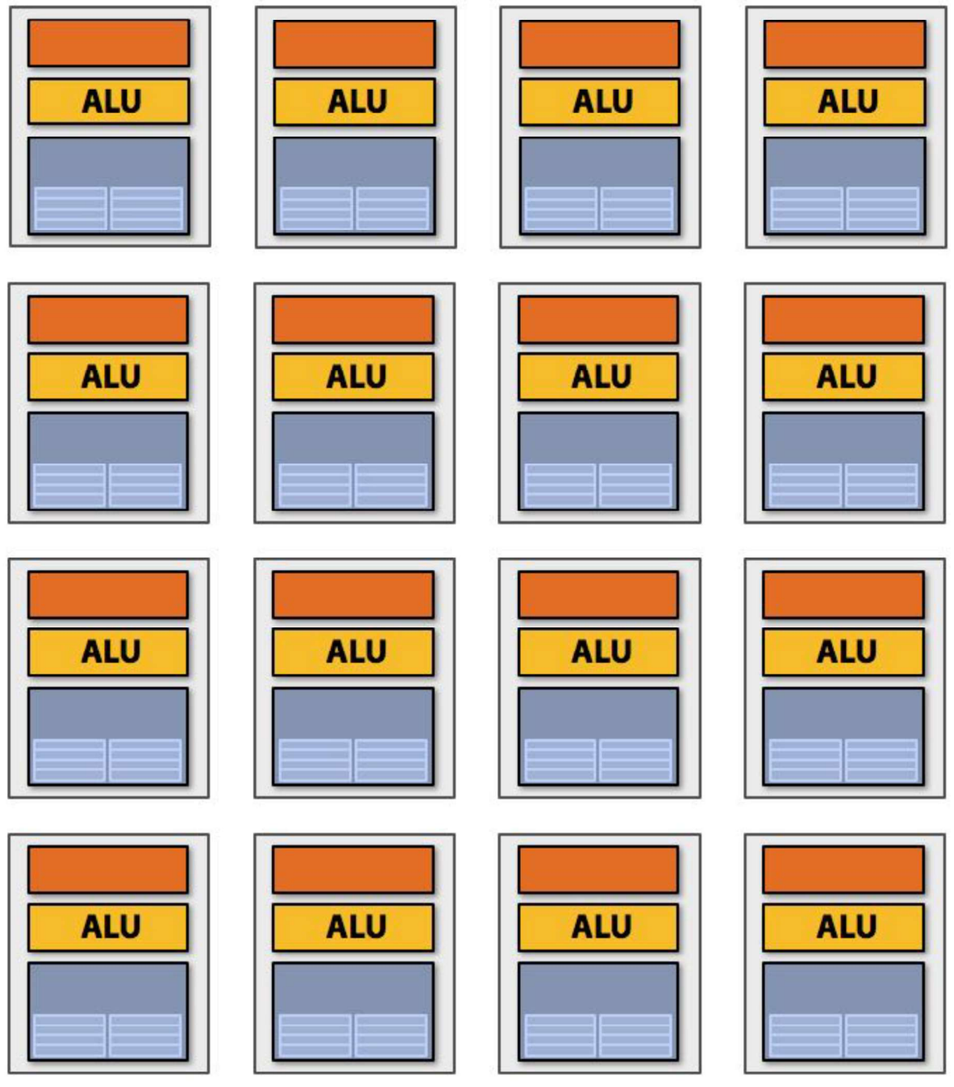
Four cores: compute four elements in parallel





Sixteen cores: compute sixteen elements in parallel

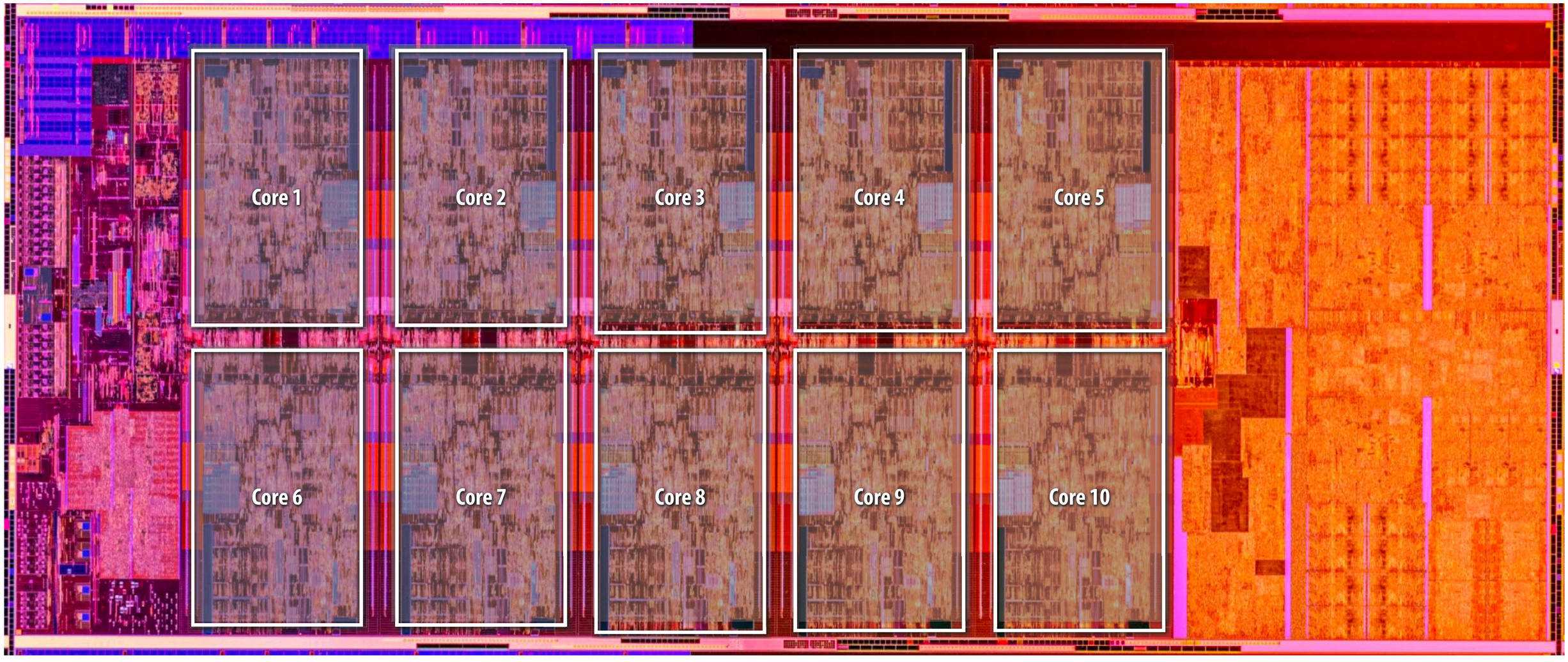




Sixteen cores, sixteen simultaneous instruction streams

Example: multi-core CPU

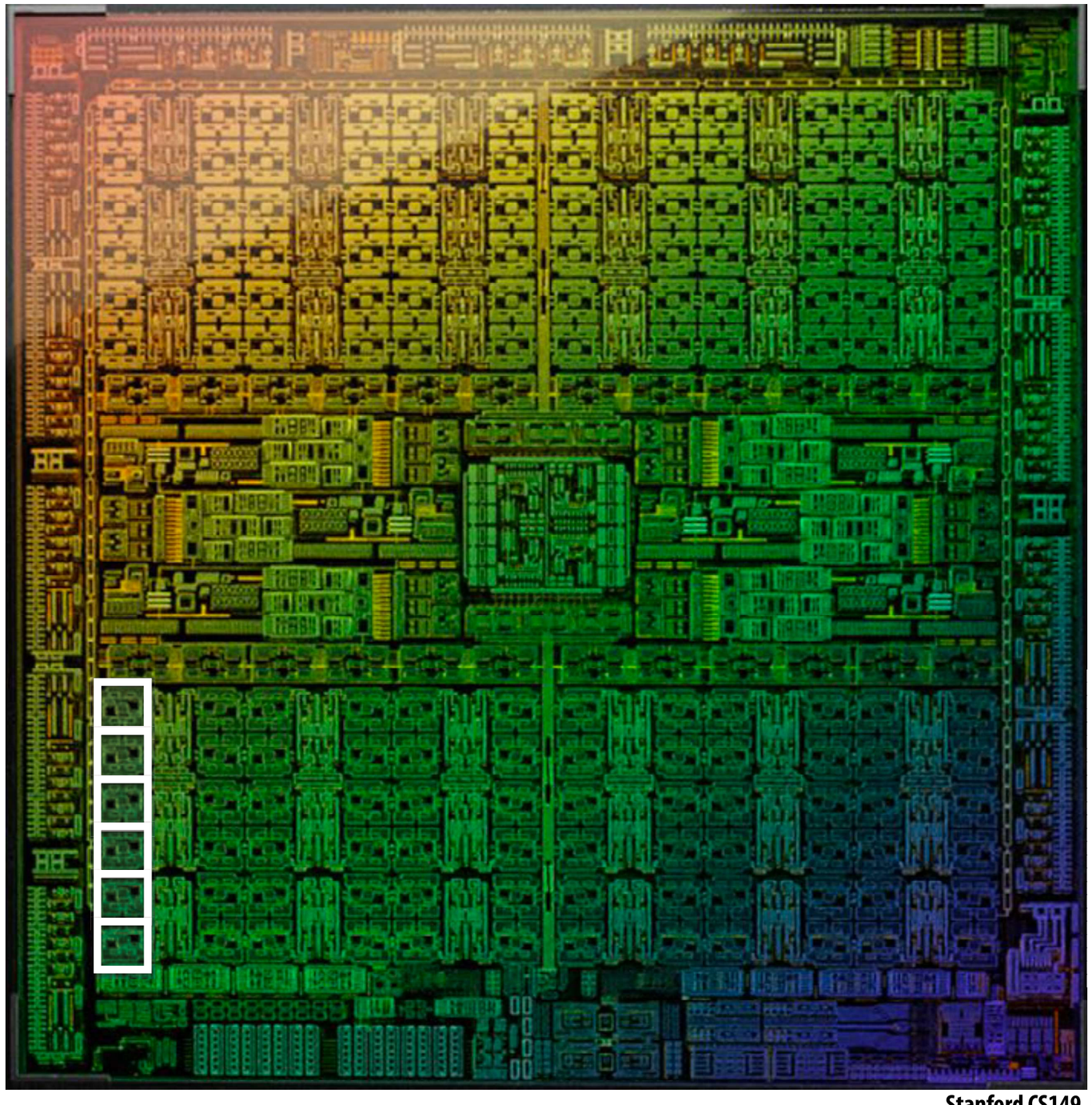
Intel "Comet Lake" 10th Generation Core i9 10-core CPU (2020)



Multi-core GPU

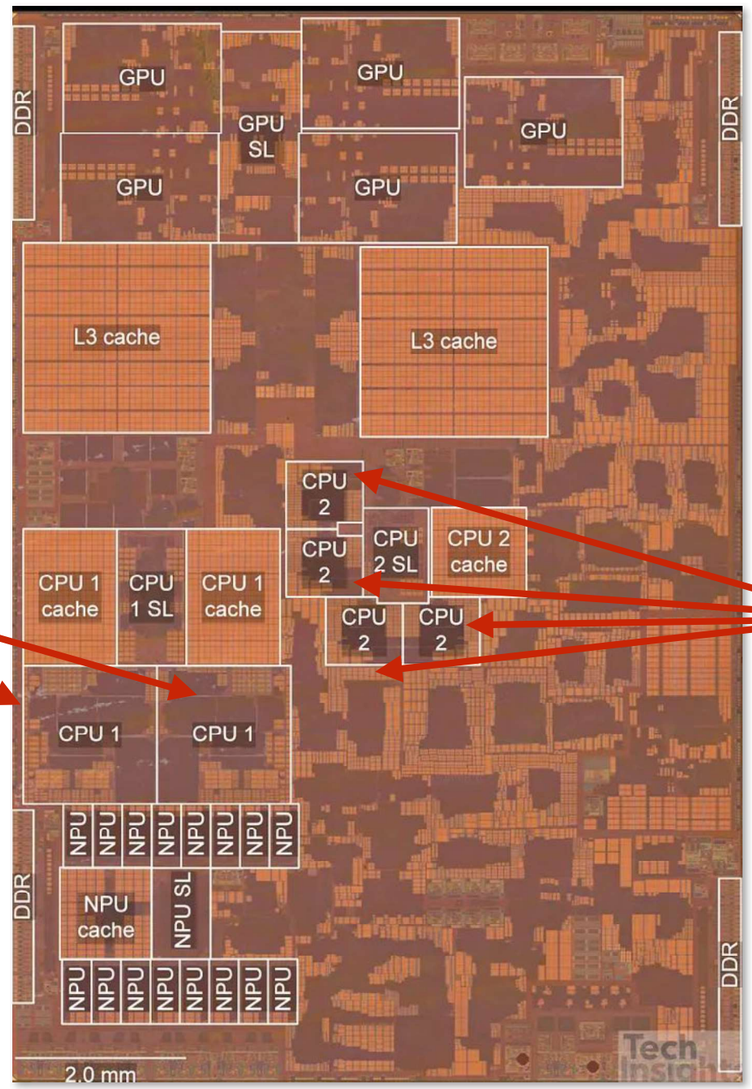
GeForce RTX 4090 (2022)

144 processing blocks (called SMs)



Apple A15 Bionic

Two “big cores” + four “small” cores

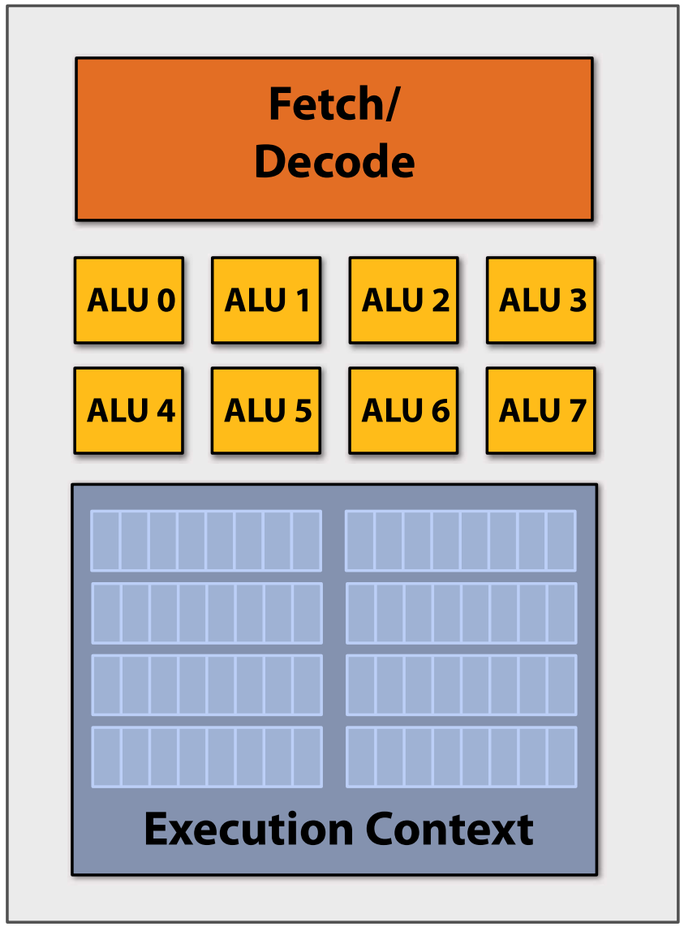


2 “big” CPU cores

4 “small” CPU cores

Image Credit: TechInsights Inc.

Add execution units (ALUs) to increase compute capability



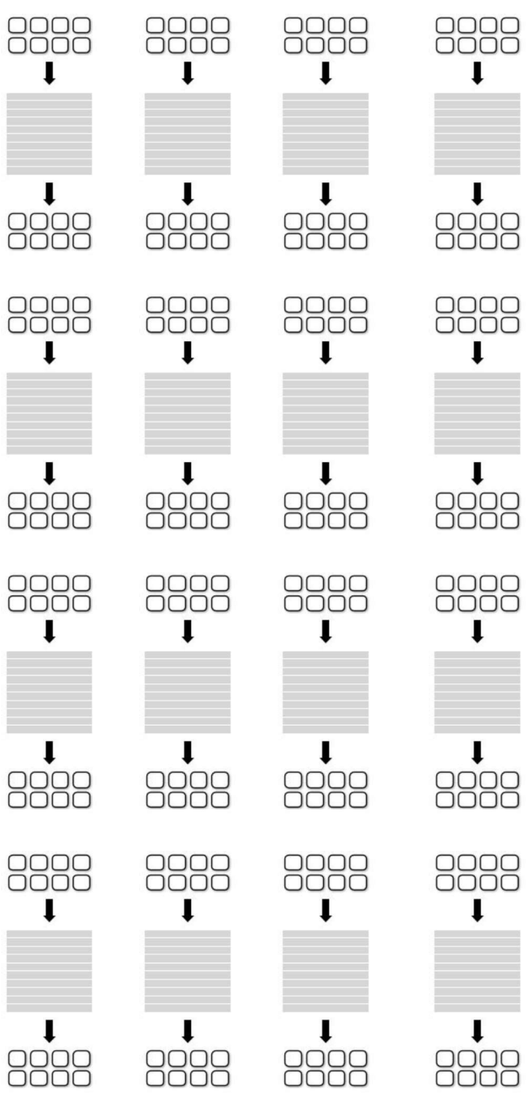
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

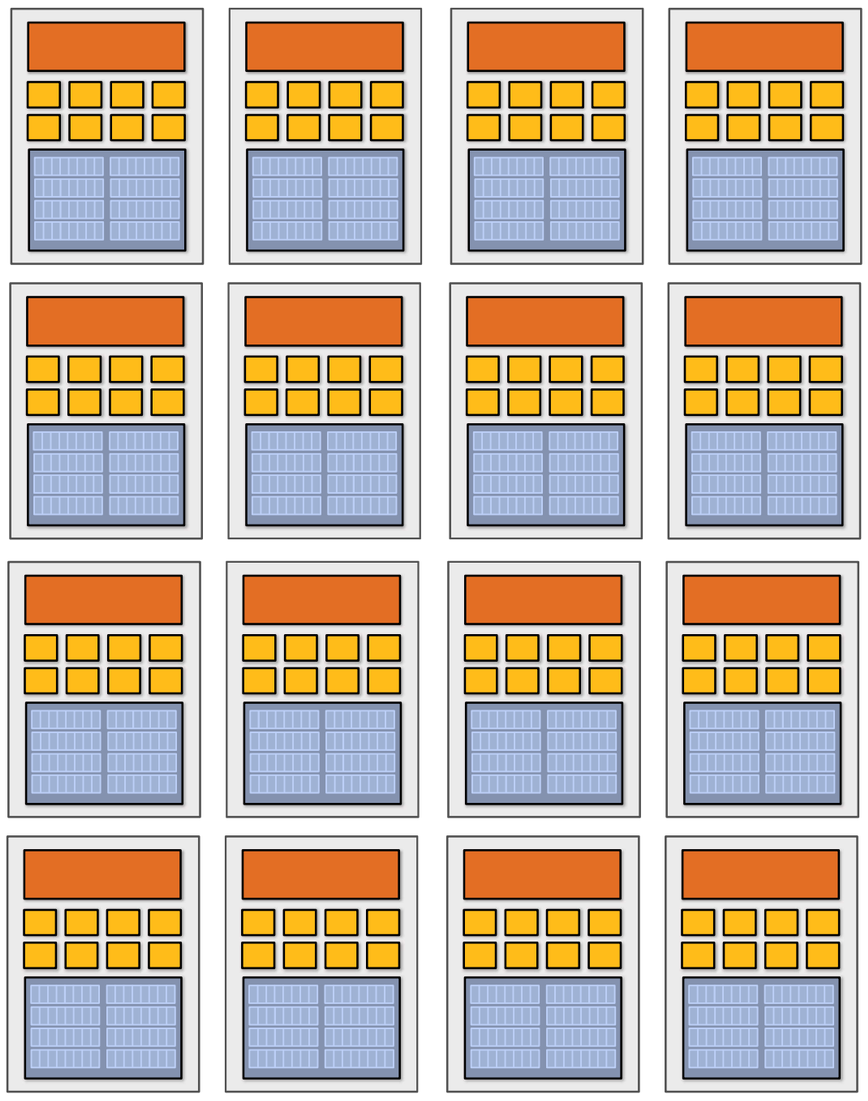
SIMD processing

Single instruction, multiple data

Same instruction broadcast to all ALUs
This operation is executed in parallel on all ALUs

16 SIMD cores: 128 elements in parallel





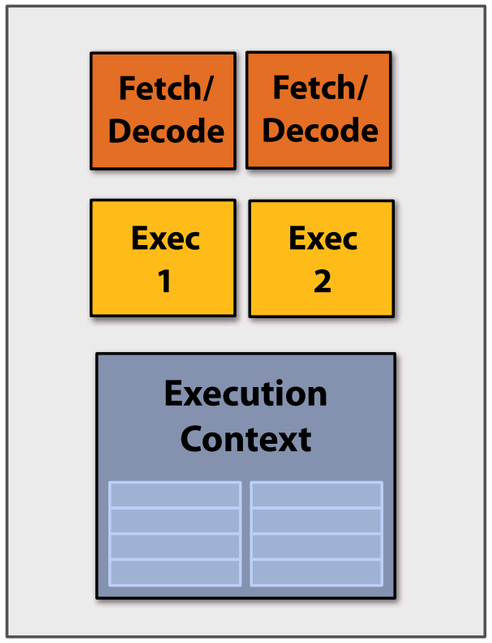
16 cores, 128 ALUs, 16 simultaneous instruction streams

Summary: three different forms of parallel execution

- **Superscalar:** exploit ILP within an instruction stream. Process different instructions from the same instruction stream in parallel (within a core)
  - Parallelism automatically discovered by the hardware during execution

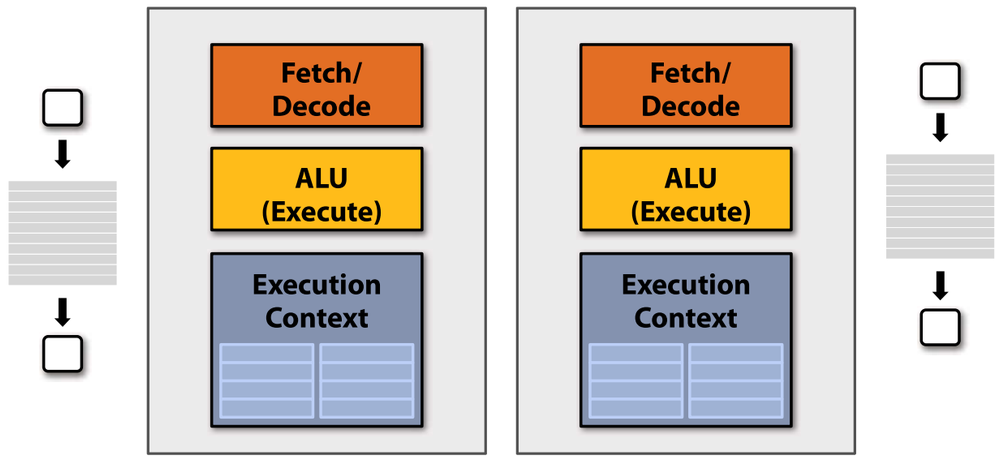
- **SIMD:** multiple ALUs controlled by same instruction (within a core)
  - Efficient for data-parallel workloads: amortize control costs over many ALUs
  - Vectorization done by compiler (explicit SIMD) or at runtime by hardware (implicit SIMD)

- **Multi-core:** use multiple processing cores
  - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
  - Software creates threads to expose parallelism to hardware (e.g., via threading API)

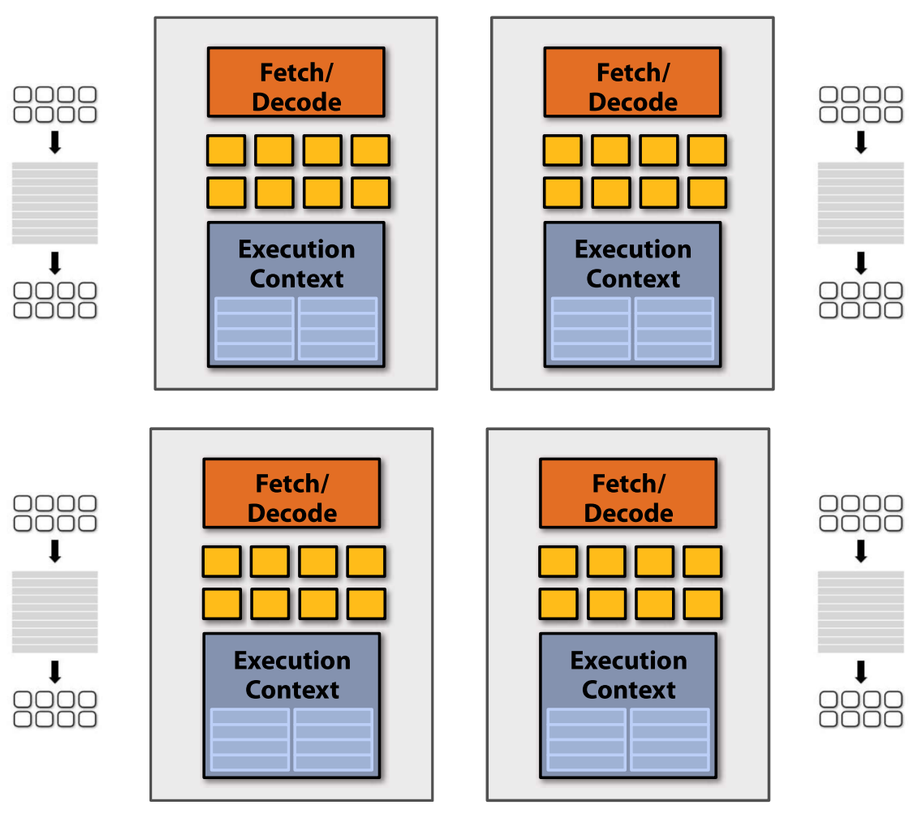


My single core, superscalar processor:
executes up to two instructions per clock from a single instruction stream (if the instructions are independent)

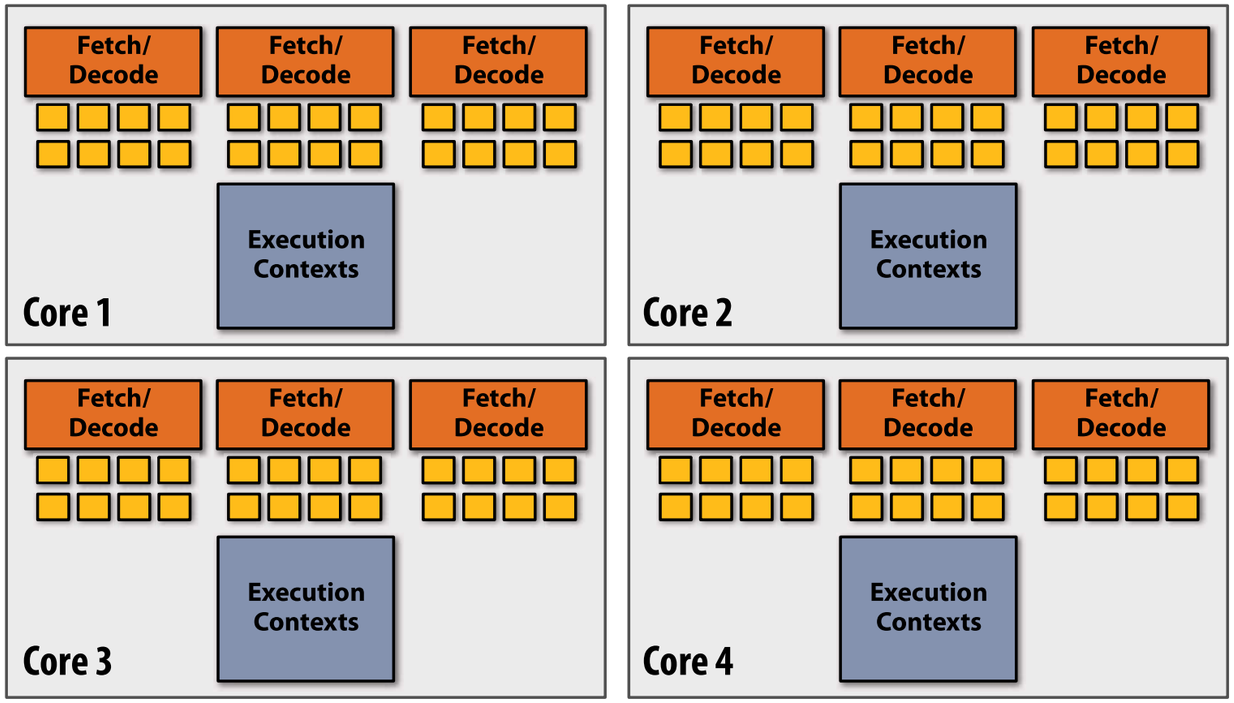
My dual-core processor:
executes one instruction per clock from one instruction stream on each core.



My SIMD quad-core processor:
executes one 8-wide SIMD instruction per clock from one instruction stream on each core.



Example: four-core Intel i7-7700K CPU (Kaby Lake)

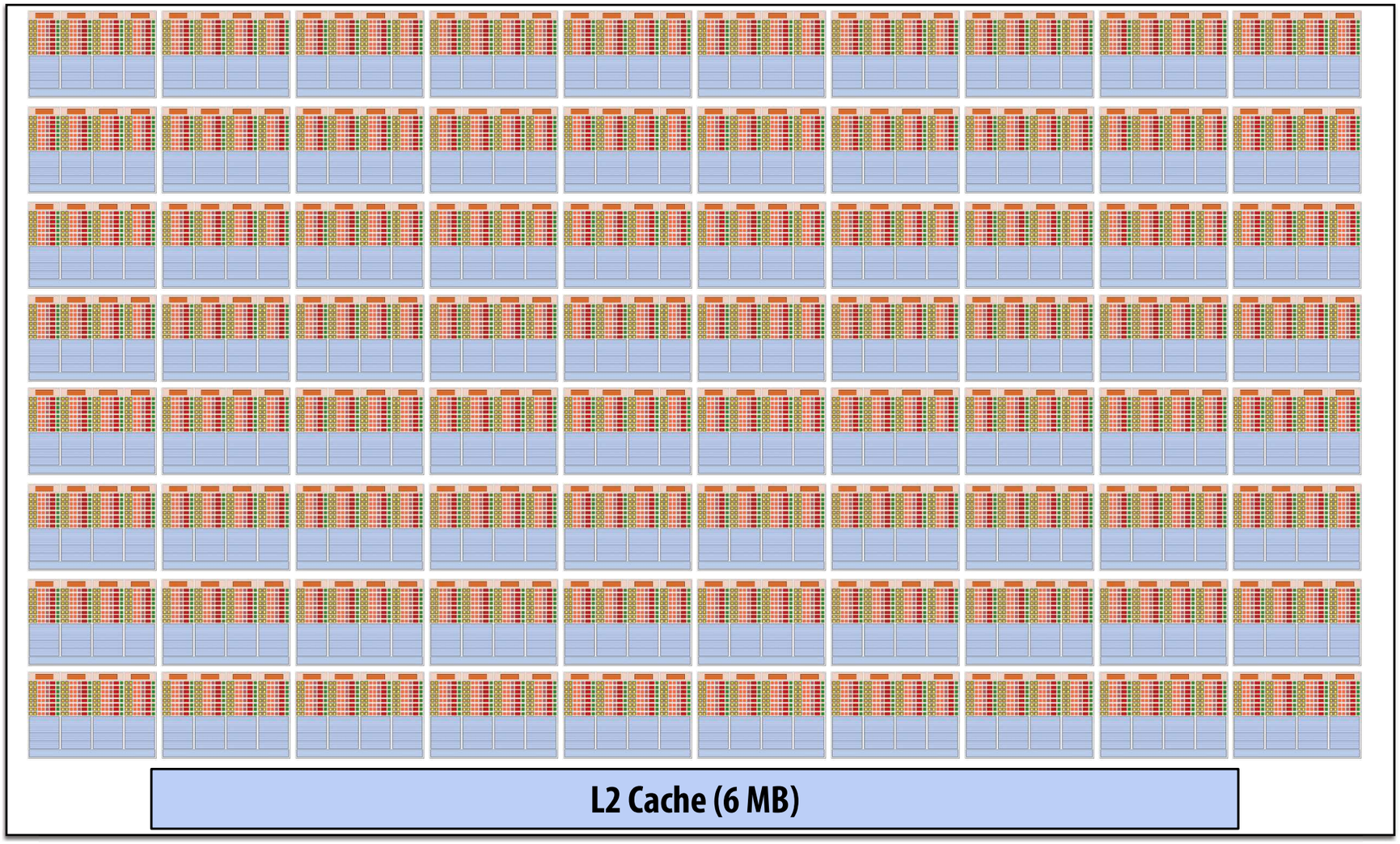


4 core processor
Three 8-wide SIMD ALUs per core
(AVX2 instructions)

4 cores x 8-wide SIMD x 3 x 4.2 GHz = 400 GFLOPs

* Showing only AVX math units, and fetch/decode unit for AVX (additional capability for integer math)

Example: NVIDIA V100 GPU



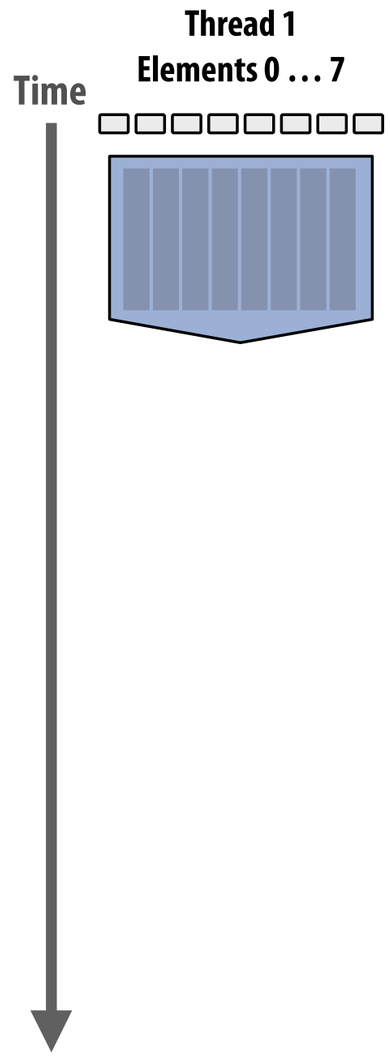
80 "SM" cores

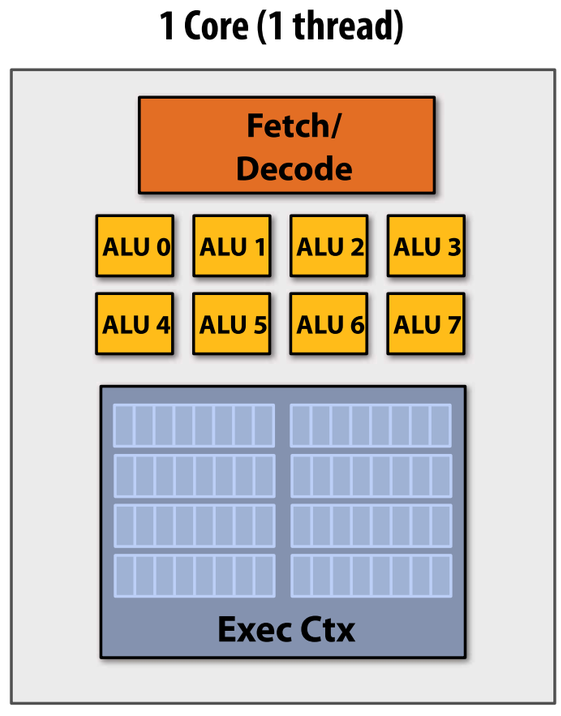
128 SIMD ALUs per "SM" (@1.6 GHz) = 16 TFLOPs (~250 Watts)

Multi-threading reduces stalls

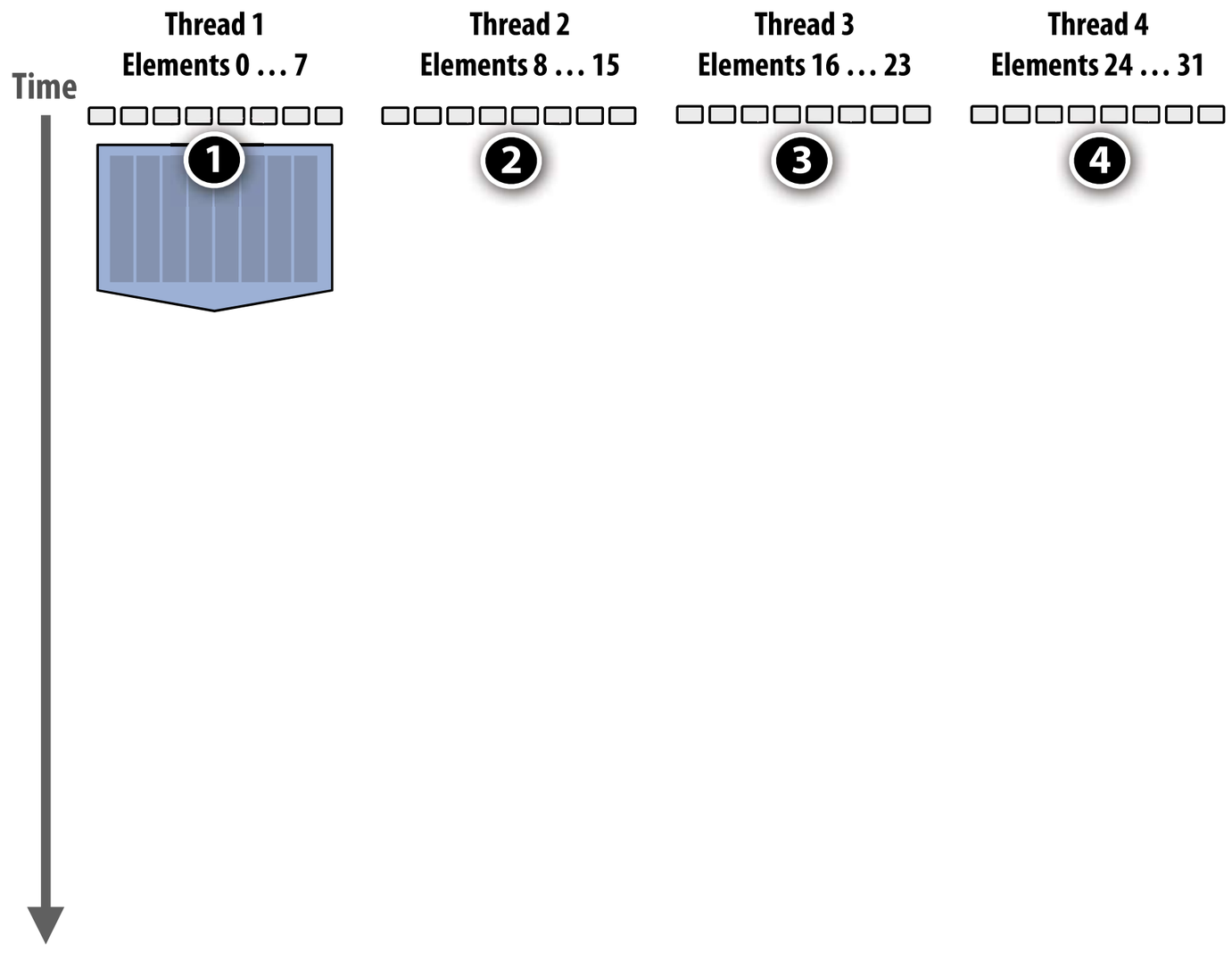
- Idea #3: interleave processing of multiple threads on the same core to hide stalls
  - If you can't make progress on the current thread… work on another one

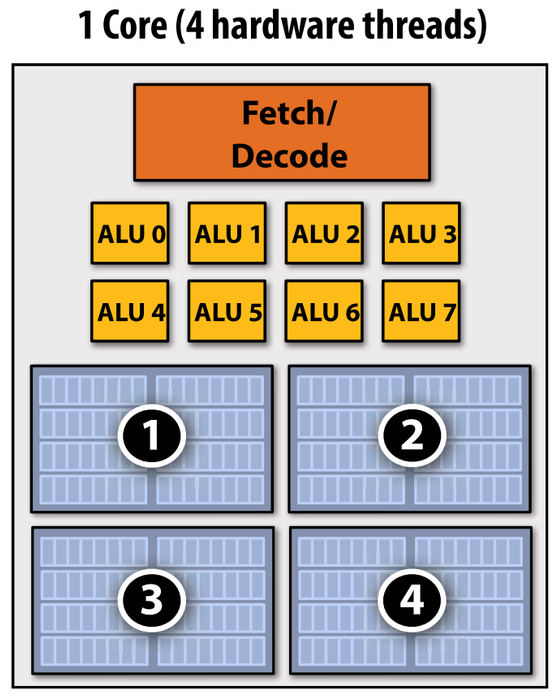
Hiding stalls with multi-threading



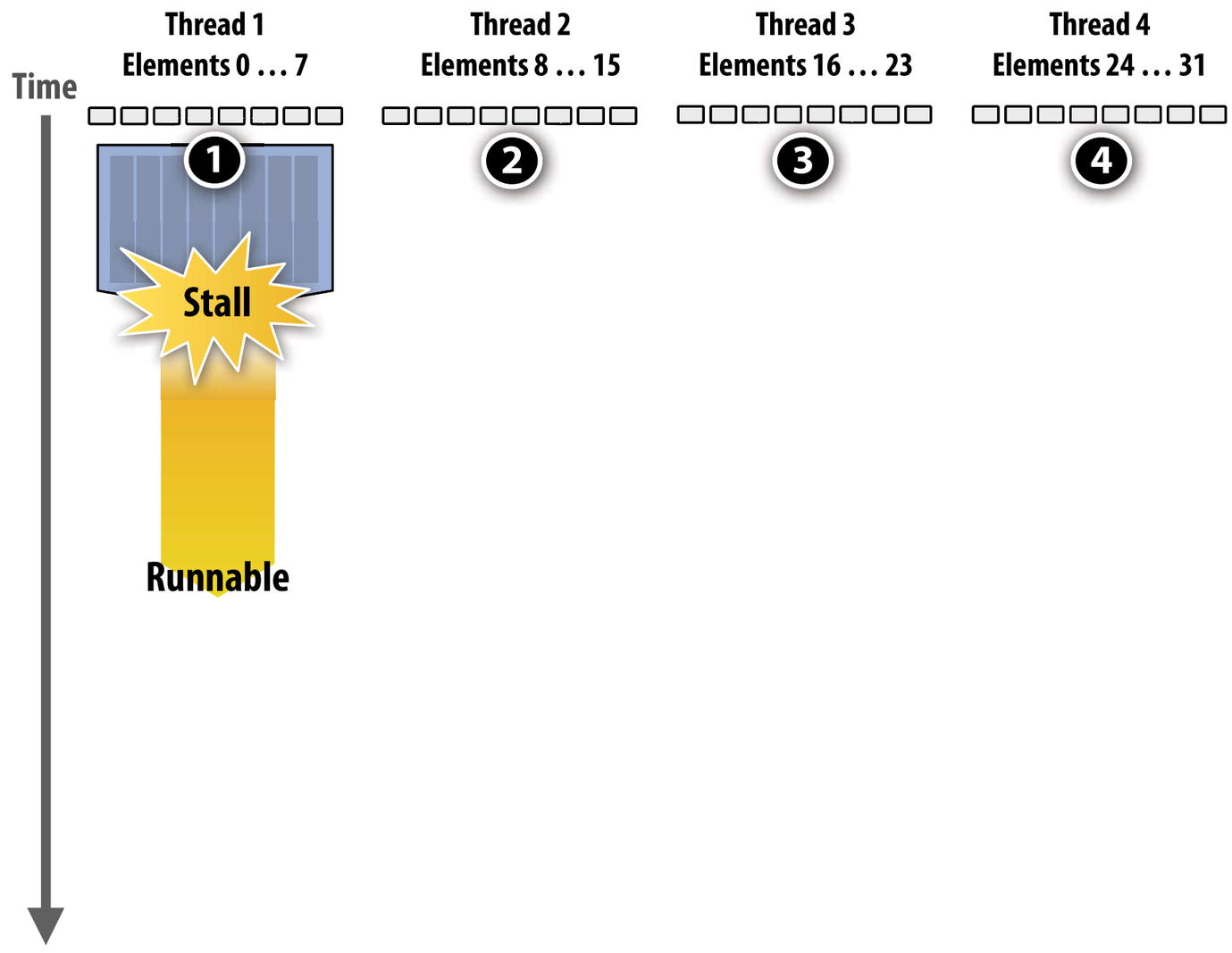


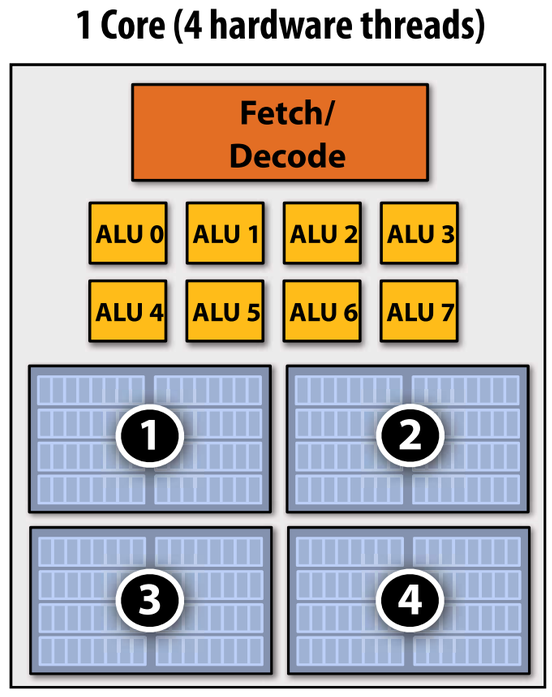
Hiding stalls with multi-threading



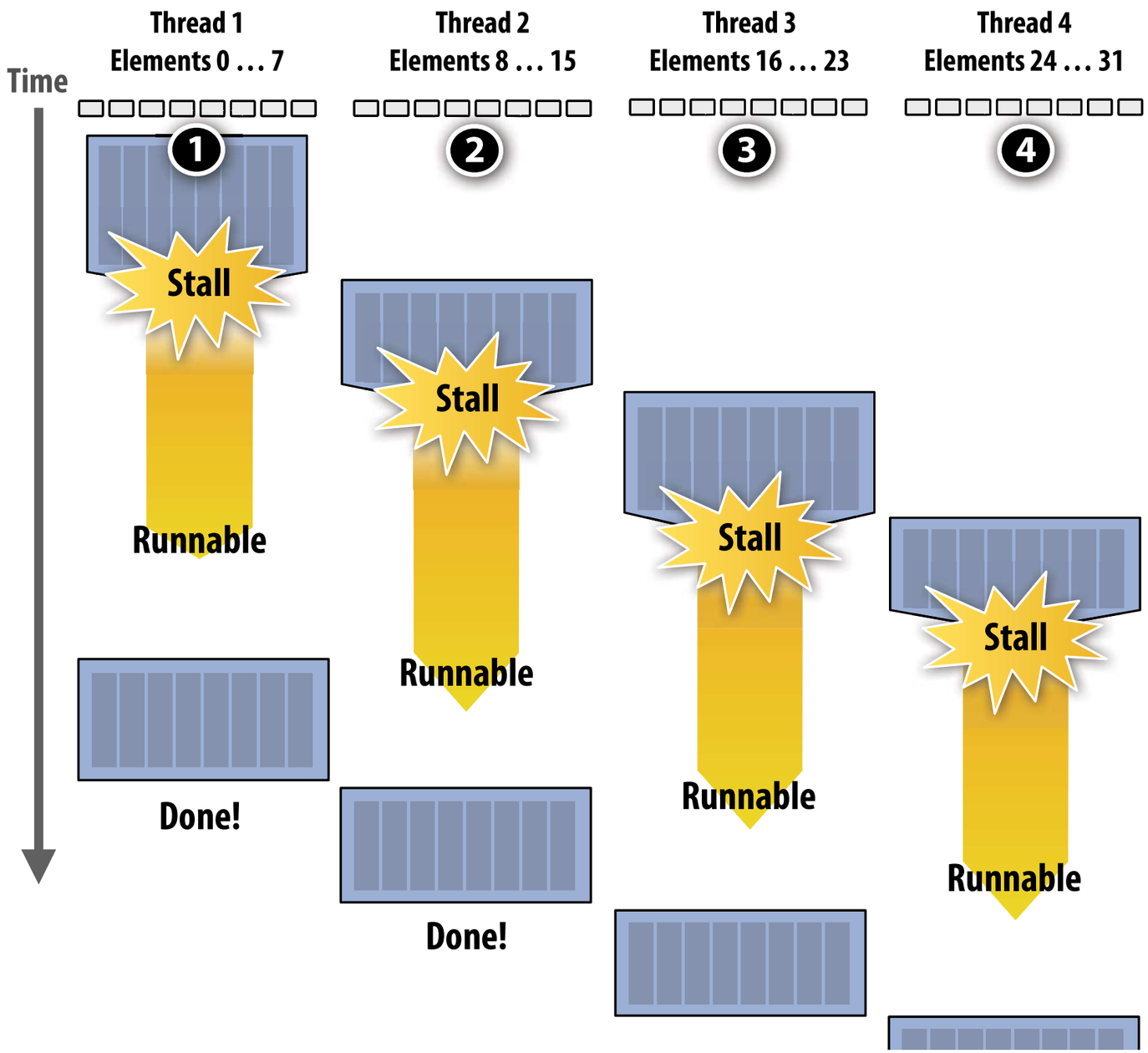


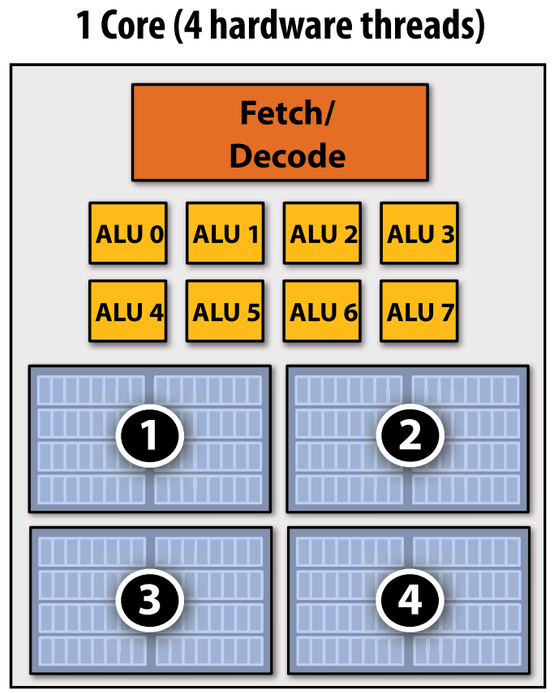
Hiding stalls with multi-threading



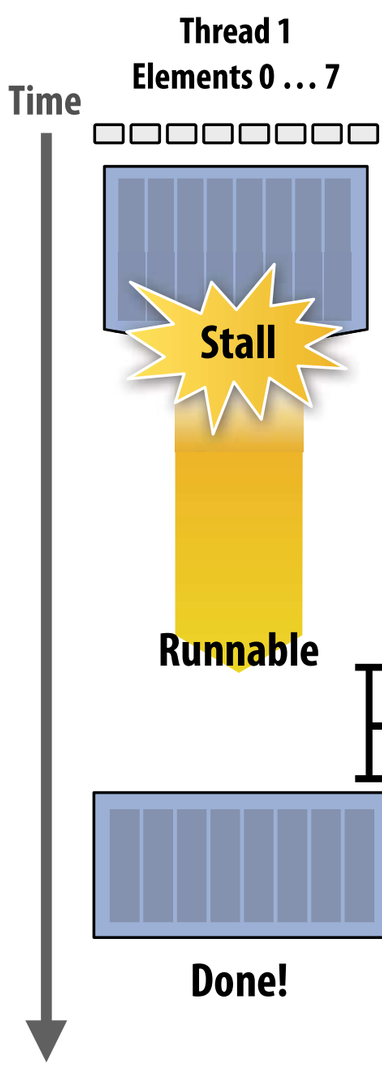


Hiding stalls with multi-threading





Throughput computing: a trade-off



Key idea of throughput-oriented systems:
Potentially increase time to complete work by any one thread, in order to increase overall system throughput when running multiple threads.

Note: during this time, this thread is runnable, but it is not being executed by the processor core.

(The core is executing instructions from another thread.)

Kayvon's fictitious multi-core chip

16 cores

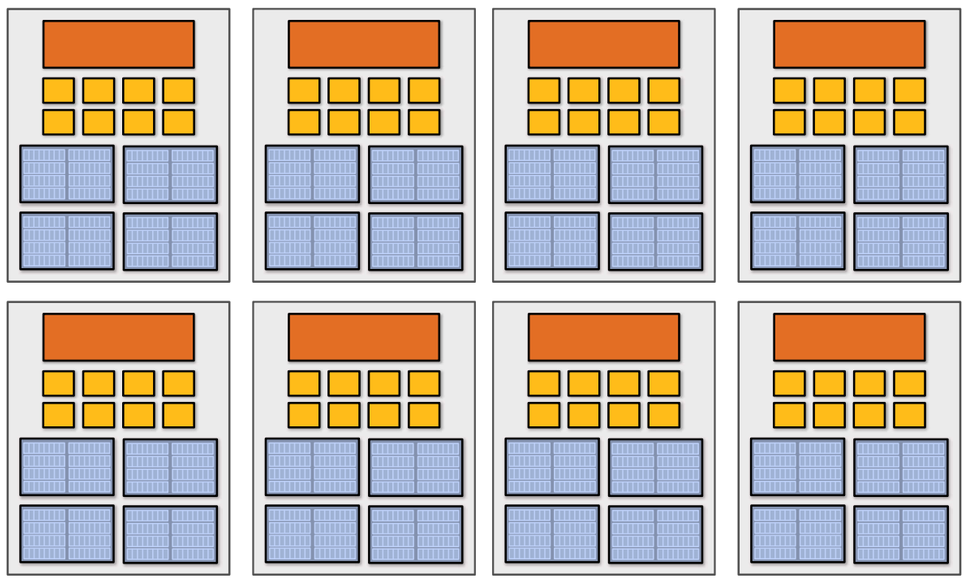
8 SIMD ALUs per core (128 total)

4 threads per core

16 simultaneous instruction streams

64 total concurrent instruction streams

512 independent pieces of work are needed to run chip with maximal latency hiding ability



The story so far…

To utilize modern parallel processors efficiently, an application must:

1. **Have sufficient parallel work to utilize all available execution units (across many cores and many execution units per core)**

2. **Groups of parallel work items must require the same sequences of instructions (to utilize SIMD execution)**

3. **Expose more parallel work than processor ALUs to enable interleaving of work to hide memory stalls**